Object-Oriented Programming

WEDNESDAY, APRIL 30

Today we'll look at how we define the abstraction boundary with our first introduction to object-oriented programming and C++ classes.

- 📚 Readings: Text 6.1-6.5
- 📝 Lecture quiz on Canvas
- 🎬 Lecture video on Canvas
- 📎 oop-geocities-quokkas.zip

Lecture Video

Click to sign in and play video

Contents

Overview: Object-Oriented Programming

Today, we delved into object-oriented programming (OOP), with a focus on classes and objects. This marks a significant transition in the course from taking a mostly client-side view of ADTs to digging into the implementation details behind the scenes and examining how we can create those ADTs in C++.

Through the rest of the quarter, we will use classes and objects as a vehicle for exploring and implementing abstractions. I want to be clear that OOP is a much broader topic than what we covered in lecture today; we dipped our toes in the waters of OOP in order to acquire a few tools we need for this quarter's journey. There is much more to OOP to be discovered on your own or through other courses.

Introduction to Classes

We started our discussion of OOP with classes, which form part of the foundation of object-oriented programming. Fundamentally, a class allows us to package together related pieces of data with functions that operate on that data.

We have already seen several classes so far this quarter, although we haven't referred to them as such. All the ADTs implemented in the Stanford C++ Libraries are classes: `Vector` , `Grid` , `Stack` , `Queue` , `Set` , and `Map` .

A `Vector` , for example, has underlying data (a collection of elements, as well as some sort of size variable that is maintained behind the scenes) as well as functions we can use to manipulate that data (`add()` , `remove()` , `insert()` , and so on).

The OOP Paradigm Shift

When OOP started taking our field by storm, it was a fundamental paradigm shift. In non-object-oriented languages, such as C, if we wanted to create a data structure, we had a data representation (typically a `struct`) that was wholly separate from the functions that operated on it. For example, if we had a vector variable, we would pass it to an `add()` function as a parameter. The add function typically was not bound to, or part of, the vector variable in any way.

Object-oriented programming breaks down a bit of the wall between data and functionality and unifies those concepts into classes in a way that jives with our notion of how we interact with many objects in the real world. For example, when I go into an elevator and press a button to take me to the fifth floor, I don't think of myself as picking up the elevator and passing it through a `goToFloor(elevator, whichFloor)` function that is wholly separate from the elevator. Rather, I think of that functionality as being built into the elevator. The button and its related functionality are fully integrated with the elevator; they are part of the fundamental identity of that elevator.

The object-oriented paradigm allows us to reflect that sort of integration in code and build programs that manipulate the state of our data in ways that really reflect our understanding of how we interact with objects in the real world.

Classes as Datatypes or Blueprints (and Other Object-Oriented Terminology)

In this section, we define the following terms:

- class
- object
- instance

When we create a new class, we are typically also creating a new **datatype**. I mentioned above that all of the ADTs we have talked about this semester are implemented as classes in the Stanford C++ Libraries. Throughout the quarter, we have already seen them used as datatypes for variable declarations. For example:

```
#include <iostream>
#include "console.h"
#include "vector.h"
using namespace std;

int main()
{
    // Below, the Vector class is being used as the datatype for variables v1 and v2.
    // v1 and v2 are both Vector objects. They are instances of the Vector class.
    Vector<int> v1;
    Vector<string> v2;

    return 0;
}
```

Insofar as a class gives us a new datatype, we can think of a class as being a sort of blueprint that tells us how to build a variable of this new type. So, in the same way that we can take the blueprints for a house and build multiple instances of that particular plan, we can take a class and build multiple variables or that type. In the example above, we have built two vectors from our blueprint. They each have their own "interiors," so to speak, which are distinct from one another; adding elements to one vector does **not** add elements to the other (in the same way that placing objects in one house does not automatically add those objects to every other house built from the same blueprint). The two vectors' interiors are even "decorated" with completely different types of: one contains ints, and the other has strings.

In the code above, we say `v1` and `v2` are **objects**. An object is an **instance** of a class. Specifically, `v1` and `v2` are instances of the `Vector` class. (Similarly, if we had the blueprints for some house called the "Monroe" model, if we built two houses from those blueprints, we would say that we had two **instances** of the Monroe model. Those houses would also be considered **objects** in the physical world.)

Why build new classes?

I mentioned today that building new classes will allow us to expand and enrich our vocabulary of abstractions, which will in turn allow us to solve more problems.

Ultimately, by understanding how classes work, we will gain the ability to add new tools to our problem-solving toolkit that we'll be able to use to solve problems that our existing ADTs might not be ideally suited for.

Earlier this quarter, I gave an example of the importance of using abstractions and how they can enhance the clarity of our code and our ability to communicate efficiently and effectively with people who are working on projects with us. In that example, I presented the following programs, both of which accomplish the same task. We saw that understanding the stack approach requires almost no effort at all for someone who knows what stacks are. With the vector approach, one has to pay attention to small, obnoxious details, like where the insertion and deletion operations are taking place, and one might then want to double-check that the code is really doing what they thought it was doing at first glance. ("This `v.remove(v.size() - 1);` operation is always removing the last element of the vector, right?") The stack abstraction saves us all that hassle.

Stack approach:

```
#include <iostream>
#include "console.h"
#include "stack.h"
using namespace std;

int main()
{
   Stack<int> s;

   s.push(10);
   s.push(20);
   s.push(15);

   cout << s.pop() << endl;
   cout << "Stack contents: " << s << endl;

   return 0;
}
```

Vector approach:

```
#include <iostream>
#include "console.h"
#include "vector.h"
using namespace std;

int main()
{
   Vector<int> v;
   v.add(10);
   v.add(20);
   v.add(15);

   cout << v.remove(v.size() - 1) << endl;
   cout << "Vector contents: " << v << endl;

   return 0;
}
```

Interface (.h) and Implementation (.cpp)

Our foray into classes marks our own paradigm shift in terms of our approach to ADTs in this class. We have so far taken a client-side approach to ADTs, where we have examined only their interfaces and had most of the implementation details for those ADTs abstracted away from us. We'll now start peeling back the curtain and looking a how we can build these data structures ourselves.

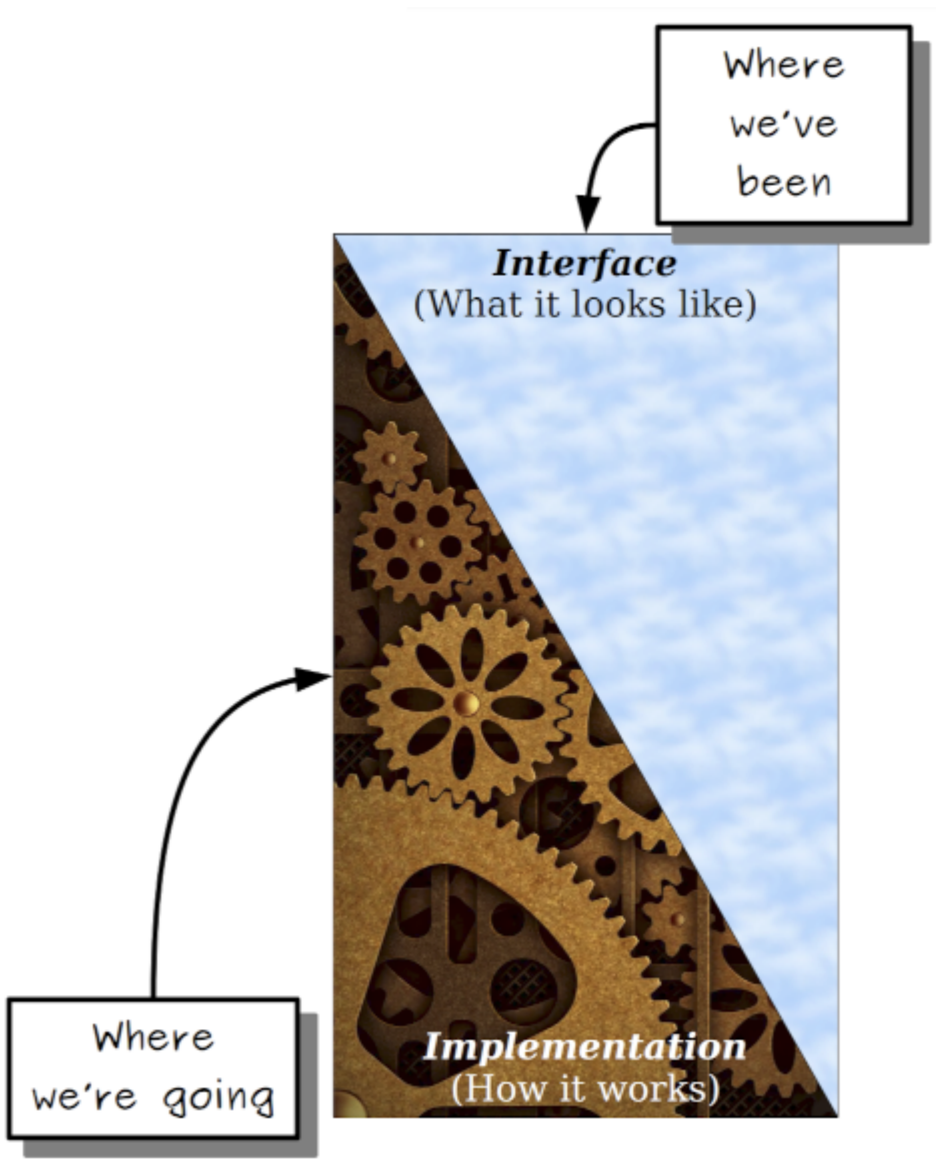


Image credit: My awesome colleague, Keith Schwarz.

When we create a class, there are two primary components we have to code up: the **interface** and the **implementation**.

- The **interface** is what a class looks like from the outside: what data is in the class and what functions we can call. On the interface side, functions are typically presented in the form of functional prototypes, not full-fledged function definitions. When you go to the Stanford C++ Library docs, you're effectively seeing the interface to each of the ADTs we've implemented there. When we create a new class, its interface will be articulated in a `.h` file (a "header file").
- The **implementation** is where we find the actual definitions of the functions that drive the behaviors of our class -- not just the functional prototypes. When we create a new class, its implementation will be articulated in a `.cpp` file (an "implementation file").

You can think of the interface as telling us "**what**" the class can do for us and the implementation as telling us "**how**" those things get done.

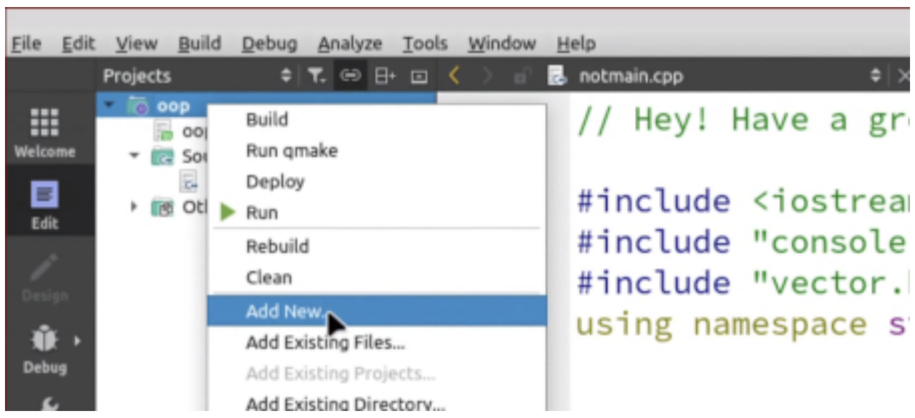
Our Goal: The Quokka Class

Throughout lecture, I referred frequently to the `Vector` class to provide grounding for my explanations of classes and objects in a class that I knew everyone was already familiar with. However, the class we implemented along the way -- which provided a more accessible example of some of these concepts -- was a `Quokka` class. Before diving into that, I introduced everyone to the following adorable creature(s):

Image credit: [Chris Hemsworth via Instagram](#).

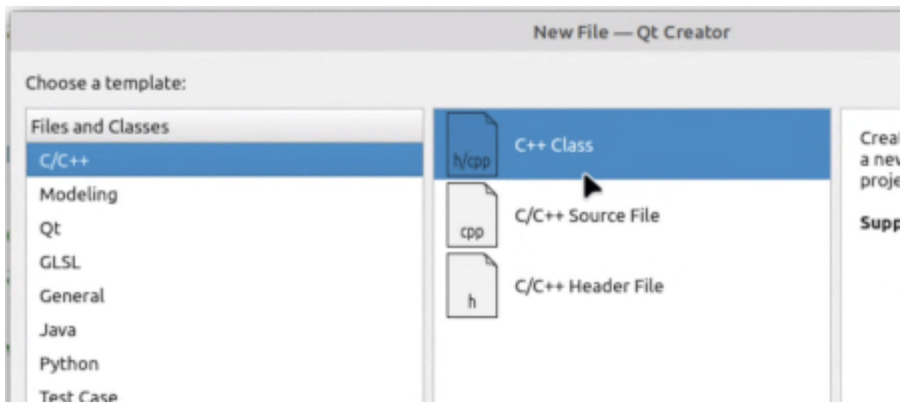
Creating a Quokka Class in the Qt Creator

Okay, here we go. To add a `Quokka` class to my existing project, I right-clicked the root folder for my project and selected "Add new..."

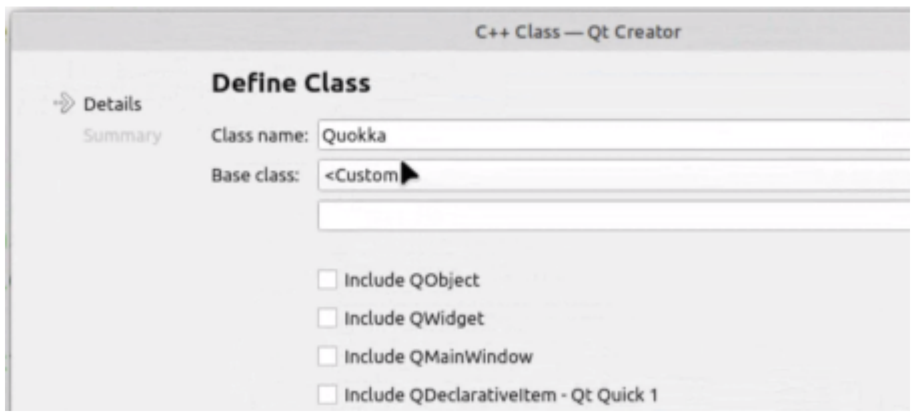


```
// Hey! Have a gr

#include <iostrea
#include "console
#include "vector.
using namespace s
```

In the menu that popped up, I selected "C++ Class." We can see from the icon in that option that we're going to get both a `.h` file and a `.cpp` file. Neat! That's what we want. We'll have a place for our class's interface and a place for its implementation.



From there, we gave our class a name:



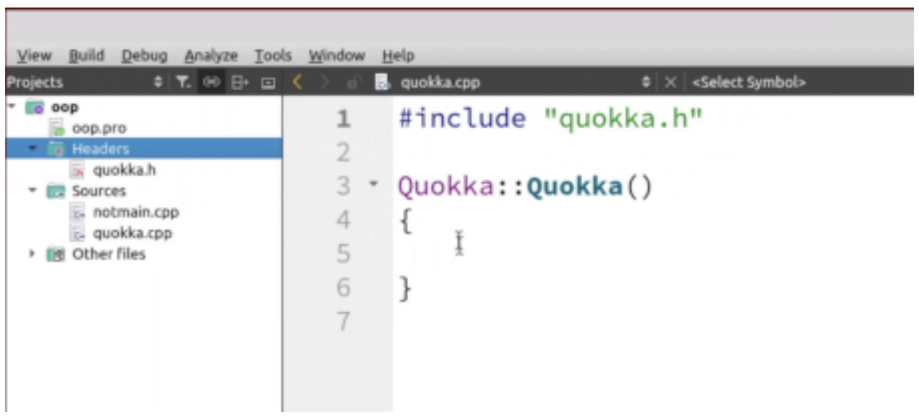
Define Class

Class name: Quokka

Base class: <Custom

(*Important note!*) A common convention in a lot of languages is to capitalize the first letter of a class name to distinguish it from variable names, function names, or primitive datatypes that are built into a language.

When we finished, Qt Creator added two files to our project: `quokka.h` and `quokka.cpp` :



```
#include "quokka.h"

Quokka::Quokka()
{

}

```

Structure of a Class's Header File (.h)

Here is the header file that was generated for us automatically:

quokka.h

```
#ifndef QUOKKA_H
#define QUOKKA_H

class Quokka
{
public:
   Quokka();
};

#endif
```

There's a lot going on there. Let's start to break it down.

First of all, Qt Creator has come in clutch and created "include guards" for us. These effectively ensure that bad things don't happen if we #include the same header file multiple times somewhere. The concept of include guards is not particularly central to our class, and so I won't elaborate beyond my comments about them in lecture, other than to point out that the format of the include guards is generally as follows:

```
#ifndef CLASS_NAME_H
#define CLASS_NAME_H

// ... more code goes here ...

#endif
```

The more interesting thing is the class definition, which follows the following basic syntax:

```
class CLASS_NAME
{
public:
   CLASS_NAME();
   // ... other variable declarations and functional prototypes go here ...
private:
   // ... other variable declarations and functional prototypes go here ...
};
```

In this class definition, we list variable declarations and the functional prototypes that will constitute our class. Under the `public:` heading, we place all the variables and functions we want any client who uses our class to have access to. Those items constitute our public-facing interface. Under the `private:` heading, we place variables and functions that we don't want our client to have access to. We mostly mark as private anything that a user could abuse to leave our class in a broken state.

(*Not mentioned in class.*) Notice that we tend not to indent `public:` and `private:` within a class definition, even though our style guide tells us to indent one level deeper any time with open a new code block. This is an exception to that rule. If we indented `public:` and `private:` , they would be the *only* lines indented just one level in that code block. All the other lines within that block would be indented *at least* two levels. We pull back `public:` and `private:` in order to cut back a bit on some unnecessary horizontal bloat in our code.

(*Not mentioned in class.*) `public` and `private` are called **access modifiers**.

(*Important note!*) Just as with structs, we need a semicolon at the end of our class definition (after the closing curly brace). If you leave off that curly brace, you might get a series of inscrutable errors at compile time.

Constructor Functions

Above, you likely noticed there's a function called `Quokka()` within our `Quokka` class definition. A function whose name matches our class name is called a **constructor function**. It is called automatically anytime we create an instance of our class, which is often useful for doing any important initialization tasks when a class is instantiated.

Structure of a Class's Implementation Source File (.cpp)

Here is the .cpp file that was generated for us automatically:

quokka.cpp

```
#include "quokka.h"

Quokka::Quokka()
{

}
```

There are a few key things to notice here:

- We `#include` our class's header file from out .cpp file. That way, this source file is aware of our class definition, including all variables and functional prototypes, and so we can freely refer to all those variables and function from any functions that we add to this file.
- The `Quokka()` constructor functino that we saw in our header file is listed here, as well.
- Before each function we define that is part of our class, we must give the class name, followed by two colons, like so: `ClassName::functionName` . This tells C++ that the function we're creating is actually part of the class in question and not some free-floating auxiliary function that exists outside of the class.

A First Draft of the Quokka Class

From there, we started fleshing out the `Quokka` class with some useful data and functionality. Here was our first draft. I have included some key notes below the code, as well:

quokka.h

```
#ifndef QUOKKA_H
#define QUOKKA_H

#include <iostream>

class Quokka
{
public:
   // member functions (which govern behaviors an object can perform)
   Quokka();  // constructor
   bool haveASnack(std::string snack);
   void printInfo();

   // member variables (which govern the state of an object)
   std::string _name;
   int _howAdorable;  // 1 through 5
   std::string _location;
};

#endif
```

quokka.cpp

```
#include <iostream>
#include "quokka.h"
using namespace std;

Quokka::Quokka()
{
}

// Recall that we need Quokka:: in front of any functions that are part of the class,
// to distinguish them from free-floating functions that exist outside the class.
// Note that the datatype comes before the class name when defining functions
// within a class.
void Quokka::printInfo()
{
   // This function can refer to all the member variables inside this class!
   cout << _name << " (how adorable: " << _howAdorable
        << ", loc: " << _location << ")" << endl;
}

// We never ended up doing anything interesting with this function, but here it is.
bool Quokka::haveASnack(string snack)
{
   return true;
}
```

main.cpp

```
#include <iostream>
#include "console.h"
#include "quokka.h"  // for Quokka class
using namespace std;

int main()
{
   // If we didn't #include "quokka.h" above, the compiler would have no idea what
   // a Quokka was when it reached the following lines, and so our program would
   // fail to compile.

   // (Terminology) Below, when we declare q1, we are doing all of the following:
   //  - creating a Quokka
   //  - instantiating the Quokka class
   //  - creating an instance of the Quokka class
   //  - creating a Quokka object

   Quokka q1;
   q1._name = "Muffinface";
   q1._howAdorable = 5;
   q1._location = "Australia";

   // q2 has its own member variables that are distinct from the member variables
   // of q1. Note that Hemmy's adorableness score is only a 4 -- possibly because
   // no matter how adorable you are, it's just hard to look like a 5/5 when you're
   // standing next to Chris Hemsworth.

   Quokka q2;
   q2._name = "Hemmy";
   q2._howAdorable = 4;
   q2._location = "Australia";

   q1.printInfo();
   q2.printInfo();

   return 0;
}
```

output

```
Muffinface (how adorable: 5, location: Australia)
Hemmy (how adorable: 4, location: Australia)
```

Terminology related to what's happening above:

- We refer to the variables within a class as its **member variables**.
- We refer to the functions within a class as its **member functions**.
- We often say that member variables define the **state** of an object, while member functions define the **behaviors** an object can perform.
- A function whose name matches the name of the class where it resides is a **constructor function**.
- (*Not mentioned in class.*) Every `Quokka` we create gets its own copies of the variables above. When that happens, we refer to those variables as **instance variables**. In OOP, we sometimes have variables that are shared across *all* instances of a class. We haven't used any of those here, and I'm not sure that we'll see any of those this quarter. You might hear me use the terms "member variables" and "instance variables"

interchangeably to refer to the sorts of variables we're defining above.

Some key notes about the code above:

- We `#include "quokka.h"` from any .cpp file where we want to refer to our `Quokka` class.
- In `quokka.h` and `quokka.cpp`, we `#include` any header files we need to enable the functionality of those files.
- As a convention, we often start member variable names with underscores. Those variables will be accessible from every function in our implementation file, and this naming convention will help us distinguish them from local variables within those functions.
- Note that it's considered best practice **not** to issue a `using namespace std;` statement in our `quokka.h` file. If we did that, then that namespace would apply to *every* source file with an `#include "quokka.h"` directive. We instead prefer to qualify anything from the `std` namespace in our header file using the tedious `std::` syntax.
- Each `Quokka` variable create has its *own* distinct copies of the variables defined in our `Quokka` class. We use the dot operator to access those member variables (as in `q1._name`).
- One of the really lovely things about the object-oriented paradigm is that when you want to know what functions you can call from an object, most IDEs will display those for you when you type the dot after your variable name (e.g., after we type `q1.` in the program above). Contrast this with trying to remember the name of a function that you want to call when you can't even quite remember what library it comes from.

Overloading Constructor Functions

The approach to creating new `Quokka` objects in the code above is really tedious. Every time we create one, we write several lines of code to give it a name, a rating of how adorable it is, and its location. We can make that process a lot less tedious by adding an overloaded constructor function to our class that takes those parameters as arguments! Here's how that's done. Changes are highlighted in peach:

quokka.h

```
#ifndef QUOKKA_H
#define QUOKKA_H

#include <iostream>

class Quokka
{
public:
   Quokka();
   Quokka(std::string name, int howAdorable, std::string profilePic);
   void printInfo();

   std::string _name;
   int _howAdorable;  // 1 through 5
   std::string _location;
   std::string _profilePic;
};

#endif
```

quokka.cpp

```
#include <iostream>
#include "quokka.h"
using namespace std;

Quokka::Quokka()
{
}

Quokka::Quokka(string name, int howAdorable, string profilePic)
{
   // This function has access to all member variables inside the class.
   _name = name;
   _howAdorable = howAdorable;
   _profilePic = profilePic;

   // Quokkas are only found (natively) in Australia, so passing this location
   // as a parameter isn't necessary.
   _location = "Australia";
}

void Quokka::printInfo()
{
   // This function can refer to all the member variables inside this class!
   cout << _name << " (how adorable: " << _howAdorable
        << ", loc: " << _location << ")" << endl;
}
```

main.cpp

```
#include <iostream>
#include "console.h"
#include "quokka.h"  // for Quokka class
using namespace std;

int main()
{
   // These now call our overloaded constructor!
   // This is so much more compact and readable!
   Quokka q1("Muffinface", 5, "muffinface.jpg");
   Quokka q2("Hemmy", 4, "hemmy.jpg");

   q1.printInfo();
   q2.printInfo();

   return 0;
}
```

output

```
Muffinface (how adorable: 5, location: Australia)
Hemmy (how adorable: 4, location: Australia)
```

Private Class Members

From there, we talked about the importance of private class members. We often make member variables or member functions private if giving a client access to those members could result in misuse that would leave an object in a broken state.

For example, the `Vector` class does not allow us to directly modify the size variable that it's keeping track of behind the scenes. Imagine how terrible it would be if we allowed that. A client could abuse it to do something like this:

```
#include <iostream>
#include "console.h"
#include "vector.h"
using namespace std;

int main()
{
   Vector<int> v;

   v.add(10);
   v.add(15);
   v.add(33);

   // THIS WOULD BE SO BAD
   v.size = 1;

   return 0;
}
```

At that point, our vector would be in a broken state. Behind the scenes, it would still have three elements, but the size would be set to 1. That would inhibit our ability to loop through all the elements in the vector, and our `remove(index)` function (which checks whether `index` is valid) would no longer be willing to access the last two elements in the vector, as it would be under the impression that they did not even exist. Yikes!

So, instead, the vector makes the size variable private. A client cannot edit the size directly, but they can still *retrieve* the size using the vector's `size()` function.

(*Super important!*) Some common reasons for making class members private include:

- Letting the client modify the member variable manually or call the member function directly could leave an object in a broken state.
- We want to implement logic that places restrictions on the values someone can place in a member variable.
- We want to control all the logic that governs changes to member variables so we can make reasonable assumptions about the state of an object throughout its entire existence.

Getters and Setters

With that in mind, we made all of our member variables in the `Quokka` class private. We then implemented various **getter** and **setter** functions:

- A **getter** is a function that simply returns the value of a private member variable.
- A **setter** is a function whose sole purpose is to change the value of a private member variable.

With the `_name` variable, we implemented both a getter and a setter. It might seem strange to make a member variable private if we want our client to be able to retrieve and modify that variable. Recall, however, that by forcing a client to use a setter -- rather than allowing them to modify a variable directly -- we can implement logic that places restrictions on the values they can set that variable to. That's exactly what we did with the setter of our `_name` variable.

Note that it's common to start the names of getter and setter functions with "get" and "set" and for the rest of the name to simply mirror the variable being accessed. (For example, `getName()` and `setName` for a `_name` variable). However, if we are only implementing a getter for some variable and no setter, we often drop the "get" and just name the function after the variable itself. That is the case with the `Vector` class's `size()` function, and that's what we did with several of the functions in our `Quokka` class below.

Our changes are as follows . As always, key changes are highlighted in peach:

quokka.h

```
#ifndef QUOKKA_H
#define QUOKKA_H

#include <iostream>

class Quokka
{
public:
   Quokka();
   Quokka(std::string name, int howAdorable, std::string profilePic);
   void printInfo();
   std::string getName();
   void setName(std::string name);
   int howAdorable();
   std::string location();
   std::string profilePic();

private:
   std::string _name;
   int _howAdorable;  // 1 through 5
   std::string _location;
   std::string _profilePic;
};

#endif
```

quokka.cpp

```
#include <iostream>
#include "lexicon.h"
#include "quokka.h"
#include "strlib.h"
using namespace std;

Quokka::Quokka()
{
}

Quokka::Quokka(string name, int howAdorable, string profilePic)
{
   _name = name;
   _howAdorable = howAdorable;
   _profilePic = profilePic;

   _location = "Australia";
}

string Quokka::getName()
{
   return _name;
}

void Quokka::setName(string name)
{
   Lexicon lex("bad_words.txt");

   for (string naughtyWord : lex)
   {
      if (stringContains(toLowerCase(name), toLowerCase(naughtyWord)))
      {
         error("Name contains bad word: " + naughtyWord);
      }
   }

   _name = name;
}

int Quokka::howAdorable()
{
   return _howAdorable;
}

string Quokka::location()
{
   return _location;
}

string Quokka::profilePic()
{
   return _profilePic;
}

void Quokka::printInfo()
{
   cout << _name << " (how adorable: " << _howAdorable
        << ", loc: " << _location << ")" << endl;
}
```

main.cpp

```
#include <iostream>
#include "console.h"
#include "quokka.h"  // for Quokka class
using namespace std;

int main()
{
   Quokka q1("Muffinface", 5, "muffinface.jpg");
   Quokka q2("Hemmy", 4, "hemmy.jpg");

   q2.setName("Covfefecake");

   q1.printInfo();
   q2.printInfo();

   return 0;
}
```

bad_words.txt

```
covfefe
moist
```

output

```
*** STANFORD C++ LIBRARY
*** The oop program has terminated unexpectedly (crashed)
*** A fatal error was reported:

    Name contains bad word: covfefe

*** To get more information about a program crash,
*** run your program again under the debugger.
```

Calling Constructors without Variable Names

We then saw how we could put a bunch of quokkas into a `Vector` without giving each of them their own variable. To do that, we just call the `Quokka()` constructor directly:

main.cpp

```
#include <iostream>
#include "console.h"
#include "quokka.h"  // for Quokka class
#include "vector.h"
using namespace std;

int main()
{
   // Yes, we can create a vector of Quokka objects!
   Vector<Quokka> v;

   v.add(Quokka("Muffinface", 5, "muffinface.jpg"));
   v.add(Quokka("Hemmy", 4, "hemmy.jpg"));
   v.add(Quokka("Percival", 5, "percival.jpg"));
   v.add(Quokka("Fred", 5, "04.jpg"));
   v.add(Quokka("Lovelace", 5, "05.jpg"));
   v.add(Quokka("Night Terror", 5, "06.jpg"));
   v.add(Quokka("Glen", 5, "07.jpg"));

   for (Quokka q : v)
   {
      q.printInfo();
   }

   return 0;
}
```

output

```
Muffinface (how adorable: 5, location: Australia)
Hemmy (how adorable: 4, location: Australia)
Percival (how adorable: 5, location: Australia)
Fred (how adorable: 5, location: Australia)
Lovelace (how adorable: 5, location: Australia)
Night Terror (how adorable: 5, location: Australia)
Glen (how adorable: 5, location: Australia)
```

Destructor Functions (Local Variables Die When We Leave a Function)

When a local variable goes out of scope in C++, it dies (by which I mean that the memory it was using it marked as unused so that memory can be used for other things). That's actually great! That means that if we call a function that creates a bunch of local variables, when that function returns, those variables are no longer taking up space in memory unnecessarily. (That's partly because local variables are created on the program stack, within the stack frame for a given function call. When the call returns, the stack frame is popped, and **_*poof*_** -- all the local variables go with it.)

Just as C++ calls a constructor function when a new instance of some class is created, it automatically calls a **destructor function** when an object goes out of scope and dies. We create a destructor function much in the same way that we create a constructor function: it uses the same name as our class, except the sytnax in C++ requires that we put a tilde (`~`) in front of that name.

To demonstrate, I coded a destructor that printed a message to the screen, and I showed that when we reached the end of `main()` , all the Quokka objects died, and their constructors were called automatically. Here are the modified `quokka.h` and `quokka.cpp` files, with changes highlighted in peach. (All other files are unmodified.)

quokka.h

```
#ifndef QUOKKA_H
#define QUOKKA_H

#include <iostream>

class Quokka
{
public:
   Quokka();
   Quokka(std::string name, int howAdorable, std::string profilePic);
   ~Quokka();
   void printInfo();
   std::string getName();
   void setName(std::string name);
   int howAdorable();
   std::string location();
   std::string profilePic();

private:
   std::string _name;
   int _howAdorable;  // 1 through 5
   std::string _location;
   std::string _profilePic;
};

#endif
```

quokka.cpp

```
#include <iostream>
#include "lexicon.h"
#include "quokka.h"
#include "strlib.h"
using namespace std;

Quokka::Quokka()
{
}

Quokka::Quokka(string name, int howAdorable, string profilePic)
{
   _name = name;
   _howAdorable = howAdorable;
   _profilePic = profilePic;

   _location = "Australia";
}

Quokka::~Quokka()
{
   cout << "R.I.P. " << _name << endl;
}

string Quokka::getName()
{
   return _name;
}

void Quokka::setName(string name)
{
   Lexicon lex("bad_words.txt");

   for (string naughtyWord : lex)
   {
      if (stringContains(toLowerCase(name), toLowerCase(naughtyWord)))
      {
         error("Name contains bad word: " + naughtyWord);
      }
   }

   _name = name;
}

int Quokka::howAdorable()
{
   return _howAdorable;
}

string Quokka::location()
{
   return _location;
}

string Quokka::profilePic()
{
   return _profilePic;
}

void Quokka::printInfo()
{
   cout << _name << " (how adorable: " << _howAdorable
        << ", loc: " << _location << ")" << endl;
}
```

main.cpp

```cpp
#include <iostream>
#include "console.h"
#include "quokka.h"  // for Quokka class
#include "vector.h"
using namespace std;

int main()
{
   Vector<Quokka> v;

   v.add(Quokka("Muffinface", 5, "muffinface.jpg"));
   v.add(Quokka("Hemmy", 4, "hemmy.jpg"));
   v.add(Quokka("Percival", 5, "percival.jpg"));
   v.add(Quokka("Fred", 5, "04.jpg"));
   v.add(Quokka("Lovelace", 5, "05.jpg"));
   v.add(Quokka("Night Terror", 5, "06.jpg"));
   v.add(Quokka("Glen", 5, "07.jpg"));

   for (Quokka q : v)
   {
      q.printInfo();
   }

   // When we return, all our Quokka objects will be deconstructed!
   return 0;
}
```

output

```
R.I.P. Muffinface
R.I.P. Hemmy
R.I.P. Muffinface
R.I.P. Hemmy
R.I.P. Percival
R.I.P. Fred
R.I.P. Muffinface
R.I.P. Hemmy
R.I.P. Percival
R.I.P. Fred
R.I.P. Lovelace
R.I.P. Night Terror
R.I.P. Glen
Muffinface (how adorable: 5, loc: Australia)
R.I.P. Muffinface
Hemmy (how adorable: 4, loc: Australia)
R.I.P. Hemmy
Percival (how adorable: 5, loc: Australia)
R.I.P. Percival
Fred (how adorable: 5, loc: Australia)
R.I.P. Fred
Lovelace (how adorable: 5, loc: Australia)
R.I.P. Lovelace
Night Terror (how adorable: 5, loc: Australia)
R.I.P. Night Terror
Glen (how adorable: 5, loc: Australia)
R.I.P. Glen
R.I.P. Muffinface
R.I.P. Hemmy
R.I.P. Percival
R.I.P. Fred
R.I.P. Lovelace
R.I.P. Night Terror
R.I.P. Glen
```

By the way, the fact that each quokka appears to be getting deconstructed multiple times is actually an indication of the fact that there are a *lot* of copies being created of each of these quokkas as they are added to the vector. New copies are also being created every time the vector expands behind the scenes.

Geocities Rendering

Attachment: oop-geocities-quokkas.zip

Finally, just for fun, I dropped some code that displayed little Geocities-inspired profile pages for each of our quokkas. To do this, I implemented a `renderProfile()` function in the `Quokka` class. That function needed to call a prewritten `renderGeocitiesPage(quokka)` function that takes as its input a `Quokka` object. The problem is, if we are inside a function within the `Quokka` class, we have access to all the member variables, but not to the `Quokka` variable name that was created back in `main()`. From within a `Quokka` member function, if we want to refer to the `Quokka` object from which we called that function, we have to use a special keyword built into C++: `this`. (See example in the `renderProfile()` function below.)

The changes are as follows. The final version of this code, which includes the `geocities` library I created, is attached at the very top of today's notes:

quokka.h

```
#ifndef QUOKKA_H
#define QUOKKA_H

#include <iostream>

class Quokka
{
public:
   Quokka();
   Quokka(std::string name, int howAdorable, std::string profilePic);
   ~Quokka();
   void printInfo();
   std::string getName();
   void setName(std::string name);
   int howAdorable();
   std::string location();
   std::string profilePic();
   void renderProfile();

private:
   std::string _name;
   int _howAdorable;  // 1 through 5
   std::string _location;
   std::string _profilePic;
};

#endif
```

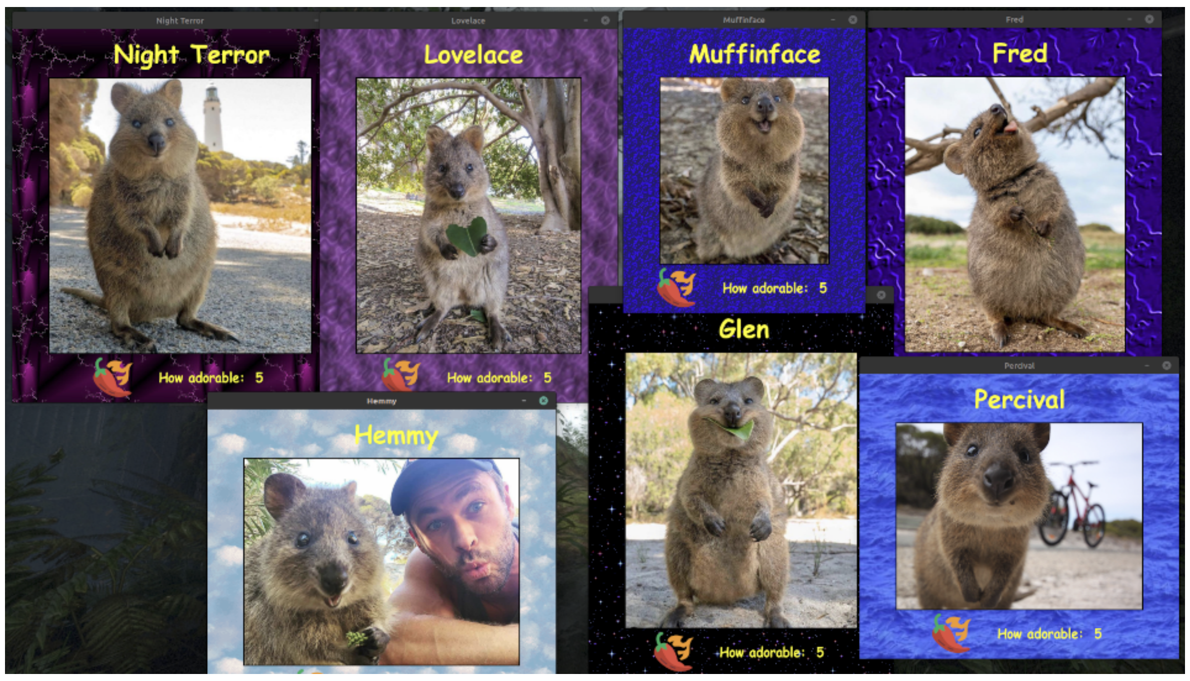
quokka.cpp

```
#include <iostream>
#include "geocities.h"
#include "lexicon.h"
#include "quokka.h"
#include "strlib.h"
using namespace std;

Quokka::Quokka()
{
}

Quokka::Quokka(string name, int howAdorable, string profilePic)
{
   _name = name;
   _howAdorable = howAdorable;
   _profilePic = profilePic;

   _location = "Australia";
}

Quokka::~Quokka()
{
   cout << "R.I.P. " << _name << endl;
}

string Quokka::getName()
{
   return _name;
}

void Quokka::setName(string name)
{
   Lexicon lex("bad_words.txt");

   for (string naughtyWord : lex)
   {
      if (stringContains(toLowerCase(name), toLowerCase(naughtyWord)))
      {
         error("Name contains bad word: " + naughtyWord);
      }
   }

   _name = name;
}

int Quokka::howAdorable()
{
   return _howAdorable;
}

string Quokka::location()
{
   return _location;
}

string Quokka::profilePic()
{
   return _profilePic;
}

void Quokka::printInfo()
{
   cout << _name << " (how adorable: " << _howAdorable
        << ", loc: " << _location << ")" << endl;
}

void Quokka::renderProfile()
{
   // The keyword 'this' refers to the object we're inside right now -- the
   // object from which we called .renderProfile(), which exists back in main().
   renderGeocitiesPage(this);
}
```

main.cpp

```
#include <iostream>
#include "console.h"
#include "quokka.h"  // for Quokka class
#include "vector.h"
using namespace std;

int main()
{
   Vector<Quokka> v;

   v.add(Quokka("Muffinface", 5, "muffinface.jpg"));
   v.add(Quokka("Hemmy", 4, "hemmy.jpg"));
   v.add(Quokka("Percival", 5, "percival.jpg"));
   v.add(Quokka("Fred", 5, "04.jpg"));
   v.add(Quokka("Lovelace", 5, "05.jpg"));
   v.add(Quokka("Night Terror", 5, "06.jpg"));
   v.add(Quokka("Glen", 5, "07.jpg"));

   for (Quokka q : v)
   {
      q.renderProfile();
   }

   return 0;
}
```

output



What's next?

On Friday, we will shift gears a bit to talk about pointers -- a topic that will enable a discussion of dynamic memory allocation next week, which will *finally* give us all the tools we need to implement complex ADTs in C++.

Exam Prep

1. Code up a basic Quokka class from scratch! You can give it whatever functionality you want, but be sure to implement the following:

- at least one constructor and one destructor function
- a few additional member functions beyond the constructor(s) and destructor(s)
- a few member variables
- a mix of public and private members
- at least one getter and at least one setter
- a `main()` function that creates `Quokka` objects and calls all the member functions you have written

2. Revisit all the notes above that are marked as "(*Not mentioned in class.*)" These notes are supplementary, but might enrich your understanding of today's material or help clarify some of the important concepts from today's lecture.

3. As always, the textbook and this week's section page are chock full of great exercises and additional examples to help reinforce this material.